ELSEVIER
Theoretical Computer Science 163 (1996) 239–243
Theoretical Computer Science

Note

Optimal heapsort algorithm

Xunrang Gu*, Yuzhang Zhu

Department of Computer Science, Shanghai University of Science and Technology, Shanghai 201800, China

Received March 1995; revised November 1995
Communicated by M. Nivat

Abstract

A new heapsort algorithm is given in this paper. Its practical value is that the efficiency of it is two times as high as that of the original algorithm in Horowitz and Sahni (1978). Also, its theoretical significance lies in the order and the main term coefficient of the complexity has optimal performance.

1. Introduction

The algorithm HEAPSORT in [4] was brought up by Williams and Floyd in 1960s, its complexity is $2n\log n + O(n)$ (All logarithms in this paper are to the base 2.). However, this algorithm is not an optimal one since the lower bound of the complexity of sorting n objects is $n\log n + O(n)$ [1, 4].

If a maximal heap (i.e. the object at each node should not be smaller than the objects as its sons) is concerned, Floyd once suggested that when the object at leaf node is small, it is appropriate to avoid those unnecessary comparisons to run in a more efficient way. But he failed to present the implementation in detail [5].

The original heapsort algorithm has been modified [2, 3, 6] on rearranging the heap to reduce the worst case time complexity and to have asymptotic optimal performance. In this paper, an optimal heapsort algorithm is presented.

2. Definitions

2.1. If the binary tree T is a heap, the subtree with the root, which can be any node in T, is called the subheap of T.

2.2. If the object attached to the node in T is removed, this node becomes vacant.

* Corresponding author.

PII S0304-3975(95)00233-2

3. Algorithms

3.1. The way to design the algorithm

The process of rearranging the heap is implemented by recursion techniques. We consider the heap with j objects, i.e., $1,2,\ldots,j$ ($j \in [3,n]$). The root object is removed and stored at position $j+1$. So we would like to rearrange $j-1$ objects at the positions $2,3,\ldots,j$ to a heap which takes the positions at $1,2,\ldots,j-1$.

Suppose the height of the current heap is h. When the root object is taken away, in order to rearrange the current heap, it is noticed that the object that can be shifted to this position (root) to become the new root object is the greater one between its leftson and its rightson. So, only by one comparison, the larger object can move up one level. Repeat this action until the vacant object appears at $\lfloor \frac{1}{2}h \rfloor$ level.

By comparing the object at position j with the object at the father's node of the vacant node, if the former is not smaller, the object at position j is shifted to this vacant position. Then it will move up to the proper position of the current heap by comparing with the object at its father's position. Otherwise, the algorithm will recursively rearrange the current subheap whose root is the vacant node. This recursive process will be carried on until $h=1$. At this moment, the object at position j will be moved to the current vacant position. Further, it takes at most two comparisons to rearrange the current subheap whose height is equal to one.

3.2. Algorithms

3.2.1. Setting up the original heap [1]

```
Procedure HEAPFIFY (i,j);
    {Arrange elements A[i]–A[j] of array A into a heap}
    if i is not a leaf
      and a son of i contains an element which is larger than i
    then begin
      Let k be a son of i with the larger element;
      interchange A[i] and A[k];
      HEAPFIFY(k,j)
    end;
Procedure BUILDHEAP;
    {Arrange elements A[1]–A[n] of array A into a heap}
    for i := ⌊½n⌋ step −1 until 1 do HEAPFIFY(i, n);
```

3.2.2. Rearranging the heap

```
Procedure REBUILDHEAP(i,j);
    begin
      if h = 1
        then   {Rearrange the current subheap whose height is one.}
```

```
        begin
          A[i] := A[j + 1];
          HEAPFIFY(i, j)
        end
      else
        begin
          count:= 1;
          while count ⩽⌊½h⌋⌋ do
              {The process of comparing the leftson with the
              rightson once and the larger moving up one level
              stops at height ⌊½h⌋ of the current subheap.}
        begin
              Let k be a son of i with the larger element;
              A[i] := A[k];
              i := k;
              count:= count + 1
        end;
      if A[j + 1]⩾A[⌊½i⌋]
      then  {A[j + 1] moves up to the proper position
            of the current subheap.}
        begin
          A[i] := A[j + 1];
          while A[i] > A[⌊½i⌋] do
            begin
              interchange A[i] and A[⌊½i⌋];
              i := ⌊½i⌋
            end
        end
      else
            begin   {Rearrange the current subheap
                        recursively.}
              h := h − ⌊½h⌋;
              REBUILDHEAP(i, j)
            end
  end
end;
```

3.2.3. Heap sorting

Input: array of elements $A[i]$ $(1 \leqslant i \leqslant n)$ to be sorted.

Output: the sorted array A.

Procedure OPTIMAL-HEAPSORT;

```
begin
    BUILDHEAP;
    for j := n step −1 until 3 do
      begin
        temp := A[1];
        h := ⌊log(j − 1)⌋;
        i := 1;
        REBUILDHEAP(i, j − 1);
        A[j] := temp
      end;
      interchange A[1] and A[2]
end;
```

4. Analysis of the complexity

We consider the objects ranging from node i to node j. Suppose h is the height of the current heap whose root is i. Therefore we have the following lemma.

Lemma 1. *The number of comparisons that the recursive procedure REBUILDHEAP takes is* $T(h) \leqslant h + \log h + 1$.

Proof. The process of rearranging the heap is correct. This can be proved by induction on the recursion depth.

When rearranging the current heap, first by comparing the leftson with the rightson of the vacant node once, the larger can move up one level. Repeat this process until it reaches at $\frac{1}{2}h$ of the current heap and totally it needs $\frac{1}{2}h$ comparisons. The next step is to compare the object $A[j+1]$ with the object at the father's node of the vacant node. At the worst case, either $A[j+1]$ moves up to the root of the current heap and it takes $\frac{1}{2}h$ comparisons, or $A[j+1]$ is searched recursively for the proper position in the current subheap whose root is the vacant node and whose height is $(1-\frac{1}{2})h = \frac{1}{2}h$. So we have the following recursion equation.

$$\begin{cases} T(h) = \frac{1}{2}h + 1 + \begin{cases} \frac{1}{2}h \\ T(\frac{1}{2}h) \end{cases} \\ T(1) \leqslant 2. \end{cases}$$

Suppose the maximum recursion depth is $K_{\max}$, so $(1/2^{K_{\max}})h = 1$, i.e., $K_{\max} = \log h$. Obviously, if the recursion depth reaches at $K_{\max}$, the number of comparisons becomes the maximum. Hence,

$$\begin{aligned} T(h) &\leqslant ((1/2)h + 1) + ((1/2^2)h + 1) + \cdots + ((1/2^{K_{\max}})h + 1) + 2 \\ &= h((1/2^1) + (1/2^2) + \cdots + (1/2^{K_{\max}})) + K_{\max} + 2 \\ &\leqslant h + \log h + 1. \end{aligned}$$

Theorem 2. *The worst case time complexity of the algorithm OPTIMAL-HEAPSORT is* $T(n) = n\log n + n\log(\log n) + O(n)$.

Proof. The correctness of the heapsort algorithm can be proved by the induction on the number of times that "FOR" loop has been executed.

The complexity of the algorithm consists of two parts:

(1) Setting up the original heap by calling BUILDHEAP; It takes time $O(n)$ [4].

(2) The time it requires for the FOR loop to perform.

Since the height of the heap with $j-1$ elements is $h = \lfloor \log(j-1) \rfloor$, so according to the above lemma, we have

$$\begin{aligned} T'(n) &\leqslant \sum_{(3 \leqslant j \leqslant n)} (\lfloor \log(j-1) \rfloor + \log \lfloor \log(j-1) \rfloor + 1) \\ &\approx n\log n + n\log(\log n) + n. \end{aligned}$$

The total complexity of OPTIMAL-HEAPSORT is $T(n) = n\log n + n\log(\log n) + O(n)$.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, MA, 1975).

[2] Gu Xunrang and Zhu Yuzhang, A New HEAPSORT algorithm and the analysis of its complexity, *Comput. J.* **33**(3) (1990) 281–282.

[3] Gu Xunrang and Zhu Yuzhang, Asymptotic optimal HEAPSORT algorithm, *Theoret. Comput. Sci.* **134** (1994) 559–565.

[4] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms* (Computer Science Press, Rock Ville, MD, 1978).

[5] D.E. Knuth, *The Art of Computer Programming*, Vol. 3, *Sorting and Searching* (Addison-Wesley, Reading, MA, 1973).

[6] I. Wegener, A simple modification of Xunrang and Yuzhang's heapsort variant improving its complexity significantly, FB Informatik, University of Dortmund, Germany, 1992.